**Модуль 12. Серіалізація та**

**копіювання об'єктів**

**Серіалізація об'єктів Python**

Серіалізація (у програмуванні) — процес перетворення будь-якої структури даних у послідовність байтів. Зворотна до операції серіалізації є операція десеріалізації (структуризації) — відновлення початкового стану структури даних із бітової послідовності.

Завжди, коли ви хочете зберегти інформацію для подальшого використання в зрозумілої машині (комп'ютеру) формі, ви робите серіалізацію. Найочевидніший приклад — це збереження даних у текстовий файл. Ви можете зберегти, наприклад, перелік витрат у текстовий файл:

expenses = {

"hotel": 150,

"breakfast": 30,

"taxi": 15,

"lunch": 20

}

file\_name = "expenses.txt"

with open(file\_name, "w") as fh:

for key, value in expenses.items():

fh.write(f"{key}|{value}\n")

Цей файл буде повністю читабельним і, якщо вам знадобиться потім завантажити цей перелік назад у Python, ви завжди зможете це зробити:

file\_name = "expenses.txt"

expenses = {}

with open(file\_name, "r") as fh:

raw\_expenses = fh.readlines()

for line in raw\_expenses:

key, value = line.split("|")

expenses[key] = int(value)

У цьому примітивному прикладі ми серіалізували та десеріалізували словник expenses.

Такий підхід цілком виправданий, особливо, якщо вам корисно зберегти людське подання інформації.

Однак, далеко не завжди такий підхід виправданий. Зауважте, що у цьому прикладі довелося винайти свій протокол серіалізації. А саме:

* символ нового рядка вказує на початок нової пари ключ-значення;
* символ | розділяє ключ і значення
* значення потрібно перетворити з рядкового представлення на числову форму.

Для роботи з даними, серіалізованими за цим протоколом, необхідно буде пам'ятати про нього та розширювати протокол за потребою.

Можна піти іншим шляхом та скористатися одним із стандартних протоколів. За такого підходу ви значно економите, використовуючи готове та протестоване рішення.

Найчастіше у Python використовуються три підходи до упакування та розпакування об'єктів:

* об'єкти, що мають рядкове представлення, приводяться до рядків і використовуються в такому вигляді (як ми зробили у прикладі вище);
* вбудований пакет pickle дозволяє працювати з вбудованими типами (словники, списки, кортежі, рядки, множини та ін.) і навіть з нескладними класами;
* протокол JSON підтримується Python і з невеликими обмеженнями дозволяє працювати з рядками, числами, списками, кортежами та словниками.

**Серіалізація об'єктів Python за допомогою pickle**

Для серіалізації/десеріалізації об'єктів Python, коли важлива швидкість, коректність і невеликий розмір пам'яті, що використовується, найкраще підійде пакет pickle.

У пакета pickle є дві пари парних методів:

dumps запаковує в byte-рядок об'єкт, loads розпаковує з byte-рядок в об'єкт. Ці методи потрібні, коли ми хочемо контролювати, що робити з byte представленням, наприклад, відправити його мережею або прийняти з мережі.

import pickle

some\_data = {

(1, 3.5): 'tuple',

2: [1, 2, 3],

'a': {'key': 'value'}

}

byte\_string = pickle.dumps(some\_data)

unpacked = pickle.loads(byte\_string)

print(unpacked == some\_data) # True

print(unpacked is some\_data) # False

У цьому прикладі упакований у byte\_string словник some\_data розпакований в unpacked та unpacked суворо дорівнює some\_data, але це все ж таки не той самий об'єкт.

dump, load упаковує у відкритий для byte-запису файл та розпаковує із відкритого для byte-читання файлу.

import pickle

some\_data = {

(1, 3.5): 'tuple',

2: [1, 2, 3],

'a': {'key': 'value'}

}

file\_name = 'data.bin'

with open(file\_name, "wb") as fh:

pickle.dump(some\_data, fh)

with open(file\_name, "rb") as fh:

unpacked = pickle.load(fh)

print(unpacked == some\_data) # True

print(unpacked is some\_data) # False

Результат аналогічний попередньому прикладу. Головна відмінність у тому, що під час виконання цього коду в робочій папці з'явився файл data.bin.

Ви можете зберігати об'єкти для подальшого використання, але з умовою. Самі класи та функції pickle зберігати не вміє і, якщо вам потрібно розпакувати упакований об'єкт класу, то сам клас повинен бути оголошений раніше у коді.

import pickle

class Human:

def \_\_init\_\_(self, name):

self.name = name

bob = Human("Bob")

encoded\_bob = pickle.dumps(bob)

decoded\_bob = pickle.loads(encoded\_bob)

bob.name == decoded\_bob.name # True

Але, якби ви захотіли передати об'єкт bob мережею іншому комп'ютеру, який нічого не знає про клас Human, то ви отримаєте помилку. Якщо ж на обох кінцях каналу зв'язку оголошено клас Human, то такий обмін працюватиме.

**Серіалізація об'єктів Python за допомогою json**

Протокол JSON (розшифровується як JavaScript Object Notation) — дуже популярний в інтернеті протокол передачі. Цей протокол має низку переваг:

* простий, його легко реалізувати;
* читабельний;
* відносно компактний (є набагато економніші протоколи).

Перша перевага зробила JSON універсальним, будь-яка сучасна мова програмування підтримує JSON. А якщо ні, то ви самі можете легко реалізувати підтримку JSON.

Недоліки у JSON теж є:

* обмежений набір типів;
* ресурсомісткий (є й вимогливіші до ресурсів протоколи).

JSON підтримує такі типи даних:

* запис (як словник у Python), в якості ключа можуть бути тільки рядки, значення — будь-який JSON тип;
* масив (як список у Python);
* число (немає різниці між цілими або дробовими);
* літерал (True, False, None);
* рядок.

Як і у Python запис та масив можуть містити вкладені записи та/або словники будь-якої глибини вкладеності.

Варто бути обережним з конвертацією типів під час роботи з JSON у Python. Кортежі під час розпакування з JSON стають списками, ключі словника, якщо вони були числами, стають рядками ж.

Python підтримує JSON і в стандартному постачанні є пакет json, в якому є все необхідне для роботи з JSON.

dumps запаковує в byte-рядок об'єкт, loads розпаковує з byte-рядка в об'єкт. Ці методи потрібні, коли ми хочемо контролювати, що робити з byte представленням, наприклад, відправити його мережею або прийняти з мережі.

import json

some\_data = {'key': 'value', 2: [1, 2, 3], 'tuple': (5, 6), 'a': {'key': 'value'}}

byte\_string = json.dumps(some\_data)

unpacked = json.loads(byte\_string)

unpacked is some\_data # False

unpacked == some\_data # False

unpacked['key'] == some\_data['key'] # True

unpacked['a'] == some\_data['a'] # True

unpacked['2'] == some\_data[2] # True

unpacked['tuple'] == [5, 6] # True

В цьому прикладі запакований в byte\_string словник some\_data розпакований в unpacked та unpacked не дорівнює some\_data, але це все ж таки не той самий об'єкт.

Окрім того:

* ключ 2 був неявно перетворений на '2';
* кортеж (5, 6) — у список [5, 6]

dump, load упаковує у відкритий для byte-запису файл та розпаковує із відкритого для byte-читання файлу.

import json

some\_data = {'key': 'value', 2: [1, 2, 3], 'tuple': (5, 6), 'a': {'key': 'value'}}

file\_name = 'data.json'

with open(file\_name, "w") as fh:

json.dump(some\_data, fh)

with open(file\_name, "r") as fh:

unpacked = json.load(fh)

unpacked is some\_data # False

unpacked == some\_data # False

unpacked['key'] == some\_data['key'] # True

unpacked['a'] == some\_data['a'] # True

unpacked['2'] == some\_data[2] # True

unpacked['tuple'] == [5, 6] # True

Результат аналогічний попередньому прикладу. Головна відмінність у тому, що під час виконання цього коду в робочій папці з'явився файл data.json.

**Робота з таблицями CSV у Python**

Ще один формат обміну інформацією, що дуже часто використовується — це табличне представлення. Відкритий формат для зберігання табличних даних, який підтримується будь-яким редактором — це формат csv. Формат csv є, по суті, тим самим текстовим файлом, але з умовою, що вся інформація в ньому розділена на колонки та рядки символами роздільниками. За замовчуванням колонки розділяють комою, а рядки — символом нового рядка. Але можна використати будь-яку іншу комбінацію символів.

Python підтримує роботу з табличними даними у форматі csv. Для цього у стандартному постачанні йде пакет csv.

import csv

with open('eggs.csv', 'w', newline='') as fh:

spam\_writer = csv.writer(fh)

spam\_writer.writerow(['Spam'] \* 5 + ['Baked Beans'])

spam\_writer.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])

with open('eggs.csv', newline='') as fh:

spam\_reader = csv.reader(fh)

for row in spam\_reader:

print(', '.join(row))

В результаті виконання цього коду в робочій папці з'явився файл *eggs.csv*. Якщо відкриєте його табличним редактором, він відкриється як таблиця.

Є два допоміжні класи в пакеті csv, які виконують роботу з табличними даними трохи зручніше:

import csv

with open('names.csv', 'w', newline='') as fh:

field\_names = ['first\_name', 'last\_name']

writer = csv.DictWriter(fh, fieldnames=field\_names)

writer.writeheader()

writer.writerow({'first\_name': 'Baked', 'last\_name': 'Beans'})

writer.writerow({'first\_name': 'Lovely', 'last\_name': 'Spam'})

writer.writerow({'first\_name': 'Wonderful', 'last\_name': 'Spam'})

with open('names.csv', newline='') as fh:

reader = csv.DictReader(fh)

for row in reader:

print(row['first\_name'], row['last\_name'])

Класи DictWriter та DictReader дозволяють працювати з рядками таблиці як зі словниками, де як ключі використовуються назви колонок (перший рядок).

Таким чином за допомогою Python можна генерувати табличні дані та імпортувати дані з таблиць.

**Управління порядком серіалізації**

Не всі об'єкти Python можна серіалізувати. Наприклад, не можна серіалізувати файловий дескриптор або системний ресурс. Але що робити, коли у вас є клас, об'єкт якого ви хочете запакувати, використовуючи pickle, але у нього є атрибути, що не серіалізуються? У такій ситуації ви можете скористатися магічними методами, які управляють серіалізацією та десеріалізацією за допомогою pickle.

Магічний метод \_\_getstate\_\_ викликається, коли pickle намагається отримати представлення об'єкта у вигляді byte-рядка. У звичайній реалізації \_\_getstate\_\_ повертає \_\_dict\_\_ словник, де зберігаються всі атрибути класу.

Але ви можете змінити цей метод.

import pickle

class Reader:

def \_\_init\_\_(self, file):

self.file = file

self.fh = open(self.file)

self.position = 0

def close(self):

self.fh.close()

def read(self, size=1):

data = self.fh.read(size)

self.position = self.fh.tell()

return data

def \_\_getstate\_\_(self):

attributes = self.\_\_dict\_\_.copy()

attributes['fh'] = None

return attributes

У цьому прикладі клас Reader можна серіалізувати, помилки через неможливість упакувати файловий дескриптор не виникне.

Цей магічний метод отримує на вхід словник, розпакований з файлу або byte-рядка. Поведінка за замовчуванням — це записати отримане значення в self.\_\_dict\_\_. Доопрацюймо клас Reader так, щоб він міг після розпакування продовжити читання з того самого місця.

import pickle

class Reader:

def \_\_init\_\_(self, file):

self.file = file

self.fh = open(self.file)

self.position = 0

def close(self):

self.fh.close()

def read(self, size=1):

data = self.fh.read(size)

self.position = self.fh.tell()

return data

def \_\_getstate\_\_(self):

attributes = {\*\*self.\_\_dict\_\_}

attributes['fh'] = None

return attributes

def \_\_setstate\_\_(self, value):

self.\_\_dict\_\_ = value

self.fh = open(value['file'])

self.fh.seek(value['position'])

**Створення копій об'єктів Python**

Python намагається заощаджувати пам'ять і не копіювати дані з однієї області пам'яті в іншу. Натомість інтерпретатор створює нове посилання (ще один псевдонім) на існуючий об'єкт, замість копіювання вмісту. Така поведінка може бути небажаною, наприклад:

my\_list = [1, 2, 3]

copy\_list = my\_list

copy\_list.append(4)

print(my\_list) # [1, 2, 3, 4]

Виходить, що copy\_list — це просто ще одне ім'я для того самого списку my\_list і, змінюючи copy\_list, ми змінюємо й my\_list. Це неочевидно і може збивати з пантелику.

Така поведінка може призводити до помилок, коли справа стосується типів, словників, списків, класів користувача, що змінюються. Для списків та словників можна скористатися явним копіюванням:

my\_list = [1, 2, 3]

copy\_list = my\_list[:]

copy\_list.append(4)

print(my\_list) # [1, 2, 3]

d = {1: 'a'}

d\_copy = {\*\*d}

d\_copy[2] = 'b'

print(d) # {1: 'a'}

Але з типами користувача так не зробиш. Щоб вирішити цю проблему, у Python є механізм копіювання — це функції із пакету copy.

Щоб створити "поверхневу" копію об'єкта, у пакеті copy є функція copy. Ця функція створює новий об'єкт такого самого типу і потім створює посилання на увесь вміст старого об'єкта в новий. Такий механізм досить хороший для роботи з об'єктами, де вже на першому рівні вкладеності немає змінних об'єктів, і він працює досить швидко.

Але для об'єктів із глибокою вкладеністю така функція все ж таки не дасть потрібного ефекту:

import copy

my\_list = [1, 2, {1: 'a'}]

copy\_list = copy.copy(my\_list)

copy\_list.append(4)

print(my\_list) # [1, 2, {1: 'a'}]

print(copy\_list) # [1, 2, {1: 'a'}, 4]

copy\_list[2][2] = 'b'

print(my\_list) # [1, 2, {1: 'a', 2: 'b'}]

З цього прикладу видно, що хоча copy\_list вже є новим об'єктом (не посилання на my\_list), але вкладений у нього словник з індексом 2 — це один і той самий словник і в copy\_list, і в my\_list.

Для ситуацій, коли нам потрібно, щоб на будь-якому рівні вкладеності створювалися нові об'єкти, а не посилання на існуючі, у пакеті copy є функція deepcopy. Ця функція рекурсивно створює нові об'єкти.

import copy

my\_list = [1, 2, {1: 'a'}]

copy\_list = copy.deepcopy(my\_list)

copy\_list.append(4)

print(my\_list) # [1, 2, {1: 'a'}]

print(copy\_list) # [1, 2, {1: 'a'}, 4]

copy\_list[2][2] = 'b'

print(my\_list) # [1, 2, {1: 'a'}]

Ще одна проблема вирішується за допомогою пакету copy — це копіювання об'єктів користувача. Щоб створити об'єкт, який буде коректно оброблятися функціями copy та deepcopy, ваш клас повинен реалізувати два магічних методи: \_\_copy\_\_ та \_\_deepcopy\_\_ для поверхневого та глибокого копіювання відповідно.

from copy import deepcopy, copy

class Expenses:

def \_\_init\_\_(self):

self.data = {}

self.places = []

def spent(self, place, value):

self.data[str(place)] = value

self.places.append(place)

def \_\_copy\_\_(self):

copy\_obj = Expenses()

copy\_obj.data = self.data

copy\_obj.places = self.places

return copy\_obj

def \_\_deepcopy\_\_(self, memo):

copy\_obj = Expenses()

memo[id(self)] = copy\_obj

copy\_obj.data = deepcopy(self.data, memo)

copy\_obj.places = deepcopy(self.places, memo)

return copy\_obj

e = Expenses()

e.spent('hotel', 100)

e.spent('taxi', 10)

print(e.places) # ['hotel', 'taxi']

e\_copy = copy(e)

print(e\_copy is e) # False

e\_copy.spent('bar', 30)

print(e.places) # ['hotel', 'taxi', 'bar']

e\_deep\_copy = deepcopy(e)

print(e\_deep\_copy is e) # False

e\_deep\_copy.spent(

'airport',

300

)

print(e.places) # ['hotel', 'taxi', 'bar']

print(e\_deep\_copy.places) # ['hotel', 'taxi', 'bar', 'airport']

Використовуючи методи \_\_copy\_\_ та \_\_deepcopy\_\_, ви можете керувати як саме буде створюватися копія вашого об'єкта. Метод \_\_deepcopy\_\_ обов'язково повинен приймати один аргумент — словник, в який записуються усі об'єкти, які піддаються копіюванню.

Це потрібно, щоб уникнути нескінченної рекурсії, якщо якийсь об'єкт є спільним для кількох копійованих. У такому випадку алгоритм глибокого копіювання може зайти в нескінченний цикл, копіюючи поперемінно об'єкти із посиланнями один на одного.

Словник memo зберігає як ключі id об'єктів і самі об'єкти як значення. Коли перевизначаємо як повинно відбуватися копіювання, ми можемо і не використовувати memo, якщо точно знаємо, що рекурсії не виникне.